Applied Cryptography

Week #9 Extra

Bernardo Portela and Rogério Reis

2023/2024

Important

- Your answers must **always** be accompanied by a justification. Presenting the final result (e.g. the result of a calculation) without the rationale that laid to said result will result in a grade of 0.
- Submit your answers via e-mail to *bernardo.portela@fc.up.pt*, with adequate identification of the group and its members.

Q1: Implementing Authenticated Encryption

Implement a prototype that exemplifies the behavior of an authenticated encryption scheme. Your goal is to ensure secure communications between *Alice* and *Bob*. The system must ensure the following:

- Message confidentiality. Use `AES-128-CTR` to do this
- Message authenticity. Use `HMAC-SHA256` to do this
- Protection against replay attacks. Include a sequence number to the messages sent

As such, this task requires you to implement `alice.py` and `bob.py`, communicating using authenticated encryption, supported by `gen.py`, which generates the pre-shared keys. Use Encrypt-then-MAC, meaning that you should calculate HMAC from the result of AES-CTR, and send both of these values over the network. The tasks are as follows:

1. Upon execution, `gen.py` must produce a file `pw` with two symmetric keys. One to be used for encryption, the other to be used for message authentication
2. Upon execution, `alice.py` and `bob.py` must begin by reading the file `pw` to get their keys.
3. Then, `alice.py` and `bob.py` must exchange the following messages:
   - From Alice: *"Hello Bob"*
   - From Bob: *"Hello Alice"*
   - From Alice: *"I would like to have dinner"*
   - From Bob: *"Me too. Same time, same place?*
   - From Alice: *"Sure!"*

Students are encouraged to tackle this challenge one step at a time. The suggested stages as as follows:

Part 1: Implement `gen.py` and test if `alice.py` and `bob.py` are reading the keys correctly.

Part 2: Implement the communication layer between Alice and Bob using sockets (*hint:* check out this reference), or pwntools (reference). Test if you can send bytes and if they are arriving without errors.

Part 3: Adapt the messages sent to now be the result of AES-128-CTR. See if the decryption is successful.

Part 4: Include the result of HMAC-SHA256 in the sent message. See if the authentication is successful.

Part 5: Include a sequence number in both `alice.py` and `bob.py`, and append it to the sent message. Check if everything is working.

Final: Adapt your prototype to have Alice and Bob send the specified messages. Check if everything is validated, and correctly decrypted.

Q2: Signing with RSA

Let d denote the private key and e denote the public key for RSA, m denote the message we want to sign and σ denote the produced signature. A naive way to use RSA for digital signatures is to simply *encrypt the message using the private key.* Consider the following signature scheme:

- **Sign:** $\sigma \leftarrow M^d \bmod N$
- **Verify:** Compute $M' \leftarrow \sigma^e \bmod N$. Accept if $M = M'$

Question - P1: Show how this signature can never be shown to be unforgeable, by constructing a valid signature for a message without knowledge of the private key d.

Full Domain Hash (FDH) are constructions that also rely on RSA to produce digital signatures, but make use of a cryptographic hash function (H) to avoid these issues. FDH behaves as follows:

- **Sign:** Compute $h \leftarrow H(M)$, and $\sigma \leftarrow h^d \bmod N$
- **Verify:** Compute $h' \leftarrow \sigma^e \bmod N$. Accept if $H(M) = h'$

Question - P2: What properties of the hash functions are we using to ensure that the previous attack no longer works?